Introducing Distribution into a RTSJ-based Component Framework *

Michal Malohlava[1,2], Aleš Plšek[2], Frédéric Loiret[2], Philippe Merle[2], Lionel Seinturier[2]

[1]Department of Software Engineering
Faculty of Mathematics and Physics
Charles University, Malostranské naměstí 25
Prague 1, 11800, Czech Republic

[2]INRIA Lille - Nord Europe, ADAM project-team
USTL-LIFL CNRS UMR 8022
Haute Borne, 40, avenue Halley
59650 Villeneuve d'Ascq, France

michal.malohlava@dsrg.mff.cuni.cz
{*ales.plsek* | *frederic.loiret* | *philippe.merle* | *lionel.seinturier*}*@inria.fr*

Abstract

The Real-Time Specification for Java (RTSJ) [6] is becoming a popular choice in the world of real-time and embedded systems. But, a growing complexness of these systems brings a demand for their distribution. However, there are only a few projects addressing application of RTSJ in distributed environments. In this paper we introduce our approach based on software connectors to support distribution in a RTSJ-based framework [9]. We propose extensions of our Soleil framework to achieve distribution while still preserving its original benefits: separation of concerns and mitigation of complexities in the system development lifecycle.

1 Introduction

An upcoming era of massively developed real-time systems brings a challenge of developing large-scale, heterogeneous and distributed systems with variously stringent QoS demands. To keep a complexity of such systems at reasonable levels, emerging solutions in this area are recently based on RTSJ [6] since it embeds real-time properties such as predictability and determinism into a general-purpose programming language.

However, the aspect of distribution in such systems still represents a challenge and brings many open issues. The state-of-the-art of distributed and real-time Java lies at its very beginning. A few proposals introducing specifications, profiles or frameworks [2, 12, 11] have been conducted, however, there is still a need of a comprehensive solution proposing a full-fledged approach that would mitigate complexities of real-time programming in distributed systems.

In our previous work [9], we have proposed a component framework for development of RTSJ-based systems. The framework provides a continuum between design and implementation of such systems and offloads burdens from developers by automatically generating an execution infrastructure of RTSJ-based systems. Nevertheless, the aspect of distribution have not been addressed there. We however envisage that supporting development of distributed real-time systems is a highly desired feature, therefore as the key contribution of this work we focus on extensions of our framework towards distribution support.

2 Related Work

The research area of distributed programming in the scope of real-time Java includes several research directions. The leading initiative is represented by an integration of Remote Method Invocation (RMI) into the RTSJ [12] and solving the task related issues such as handling real-time properties [4, 12] or memory allocation [3, 4]. The results of these projects are reflected in a status report of *JSR 50* [2] which tries to cover all aspects of distribution (real-time properties handing, failure semantics, distributed threads and their scheduling). A similar approach proposes a profile for distributed hard real-time programming [11], however, a framework addressing a comprehensively challenge of developing such a complex system still has not been proposed.

Another research area covers the *Real-time CORBA specification*,[1] which can serve as a particular base for a requirements analysis of real-time distributed systems. Its main implementor in the RTSJ world is RTZen [10]. Although it is a middleware implementing almost all parts

[1]This work has been partially funded by the Czech Academy of Sciences project 1ET400300504, the ANR/RNTL Flex-eWare project, and by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy.

[1]OMG, Real-time CORBA, v1.2, `http://www.omg.org/technology/documents/formal/real-time_CORBA.htm`.

of the Real-time CORBA specification within the scope of RTSJ, it only focuses on a core of communication and does not provide any abstraction.

From our point of view, all these projects focus only on low-level communication issues and their integration into the scope of RTSJ, they do not address any higher abstraction of the real-time communication. It could however be beneficial to reflect distribution in different stages of the application lifecycle (design, implementation, runtime).

3 Challenges of Distributed RTSJ-based Designing and Programming

Integration of distribution into a real-time component framework [1] is a challenge involving an analysis of requirements dedicated to real-time systems as well as requirements coming from used RTSJ. All these requirements affect not only a way of specifying model artifacts (components, bindings and their properties) but also its runtime structure and the process of its initialization.

In this section we therefore determine a scope of requirements which have to be reflected by a distributed system within a real-time environment at all stages of the application lifecycle.

3.1 Requirements and Challenges

Real-time properties. Since real-time programming introduces specific requirements on distributed systems (e.g. priorities of running tasks, computational deadlines), they play a substantial role during their development. These properties influence remote connections and superimpose new constraints over them. Some realtime properties have to be propagated between remote parts of applications (e.g. priority of a client thread) and others have to be reflected during creation of the connection (e.g. end-to-end time).

RTSJ requirements. Moreover, employing RTSJ in development of distributed RT-Java-based systems is also affected by the particularities of its specification. It distinguishes between a heap memory and non-heap memory and specifies how they can be accessed by the different threads. These facts enforce different memory allocation ways as well as a specific utilization of schedulable entities (threads, timers, events). However, the specification silences about distribution aspects and therefore these complexities need to be resolved by the developers.

Integration level. Furthermore, the integration of distribution into a component framework yields a decision at which level of abstraction the distribution will be incorporated into the framework and how a component-application developer will manipulate with real-time properties. Whether to hide the manipulation from the developer or not.

These questions were discussed in the scope of RMI integration into RTSJ presented in [12]. It distinguishes three basic levels of RMI integration (denoted as `L0`, `L1` and `L2`) from different views.[2] `L0` is the minimal level of the integration with no support for real-time properties from underlying technology. The level `L1` requires a transparent manipulation with scheduling parameters or timing constraints and finally `L2` declares semantics for the distributed thread concept [2] which represents a fully transparent real-time programming model.

We partially adopt this idea of integration levels in our approach. The primary objective is an integration of distribution into the RTSJ-based component system at a level corresponding to `L1`. We however generalize the idea of the level `L1`, originally tightly coupled with RMI, to address the full span of possible communication middlewares (RMI, CORBA) in distributed environments. We therefore address the following contributions to meet this generic goal:

(i) **Scheduling Parameters.** To handle transparently scheduling parameters which are associated to component threads. The task involves a transportation of parameters from a client to a server where it is required, configuration of an underlying middleware (e.g. in case of CORBA, creating priority lanes), pre-reservation of connections for selected priorities, etc.;

(ii) **Determinism.** To ensure that the generated runtime infrastructure does not affect determinism and timely delivery assured by a used underlying middleware;

(iii) **RTSJ Rules and Restrictions.** To handle memory and thread differences between components and a used middleware. This also covers handling of a memory allocation of call parameters (e.g. CORBA parameter holders) and auxiliary artifacts (e.g. adaptors, call serializers),

(iv) **Communication Styles** To provide different communication styles [5, 8] which are common in the real-time and embedded systems world (synchronous and asynchronous method call, asynchronous messaging).

3.2 Goals of this Work

Our philosophy postulated in [1] states that the RTSJ concerns influence the architecture of applications and therefore must be considered at early stages of the system development lifecycle. We have applied this in [9] where we propose a framework clarifying all the steps of the system development lifecycle. In this work we follow the same principles, there are therefore two key objectives: (i) **Development Methodology** that clarifies specification of model artifacts and properties that will cover distributed real-time

[2] *Programming model* (identification of remote objects), *development tools* and *implementation model* (real-time properties transport mechanism). In our case, the first two models are realized by the component framework [1] therefore the following text is interested in the third one.

requirements and create an abstract layer which will hide low-level distribution concerns from component designers, and (ii) **Execution Infrastructure** that manages transparent deployment and run of distribution support inside the execution infrastructure.

Although distribution has to be captured at all stages of the system development lifecycle, in this paper, we focus on the design and generation of an execution infrastructure.

4 Supporting Distribution in Real-time Java

The basic idea of our approach is inspired by a solution in which components communicate through architecture-level software connectors that are implemented using a middleware [7]. This approach preserves the properties of the architecture-level connectors while leveraging the beneficial capabilities of the underlying middleware. Moreover, we integrate this approach into the *Soleil* framework proposed in [9].

Soleil is the execution infrastructure generator that generates a system's infrastructure on the basis of a given architecture. Thus we automatically obtain connector implementations, consequently mitigating complexities of the system development. Additionally, the process of designing and implementing connectors addresses the real-time challenges identified in Section 3.

From the high-level point of view, we adopt a general approach to a connector generation presented in [5], we however focus on more lightweight and especially RTSJ tightly coupled solution.

4.1 Applying Component Connectors

Design Time. At design time we perceive connectors as representations of bindings between functional components. A binding has attached non-functional properties such as benchmarking, enforcement of a dedicated connection or prescribed utilization of a given middleware. Furthermore, the binding connects components which also have associated properties (e.g. call deadlines for interface operations) or they receive derived properties from non-functional components in which they are placed (e.g. memory allocation context, thread priorities). All these properties are reflected in the connector architecture representing the binding.

The proposed connector architecture is based on a concept of *chains of interceptors* which are connected to concerned interfaces, as illustrated in Figure 1. Each interceptor in the chain symbolizes one non-functional concern which reflects communication *in situ* (e.g. monitoring) or modifies communication (e.g. adaptation of method call parameters). The presence and position of the interceptor in the chain is influenced by properties of the modeled binding and also by a presence of other interceptors.

This representation of connectors allows us to build them easily with different functionalities — by selecting relevant interceptors and their order in according to specified properties. As well as, the division of the connector architecture into separated interceptors permits handling real-time specific properties separately in dedicated interceptors. The chosen architecture also brings advantages in dealing with issues triggered by using RTSJ [1] such as memory scopes crossing or copying between memory aread.

Generation Process. The connector generation process includes:

(i) **Chain Structure Selection.** Which involves selecting interceptors and their order in according to binding properties (specified and derived) and also to RTSJ requirements, e.g. selecting memory allocation areas and adapting memory or thread differences;

(ii) **Interceptor Code Generation.** The task involves generation of interceptors and of a selected middleware specific code (e.g. initialization of middleware, setting connection parameters).

Furthermore, different optimizations in the chain or in its selected parts are possible, similarly as proposed in [9].

Runtime. The preservation of the connector architecture at the runtime level permits modification of connector attributes. Either simple attribute modifications affecting only one interceptor are possible (e.g modification of middleware threads priority) or even more advanced adaptations of the connector structures can be performed (e.g. update of interceptors in a chain, change of the interceptors order).

4.2 Illustration Example

The proposed concept was applied in an implementation of a simple example presented in [9]. Concretely, we model a realtime communication between two active components - *ProductLine* and *MonitoringSystem* allocated in a non-heap memory. Both components have associated properties defining components' thread priorities. The binding between these components is modeled as a remote binding with two associated non-functional properties — the first one enforces utilization of a distribution enabling technology (in our case we use *RTZen* middleware [10]) and the second one identifies asynchronous method call.

These simple properties involve several tasks which has to be covered by the generated connector and its *chain* architecture: (i) implementation of core distribution with help of RTZen. This also involves generation of low-level CORBA interfaces, helpers, value holders in according to

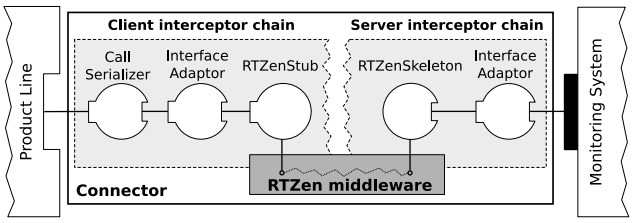

Figure 1. Structure of RTZen-based connector

a specified IDL; (ii) configuration of underlying middleware - adjustment of CORBA policies to reflect components' thread properties; (iii) asynchronous method calls in case the underlying middleware does not support them; and (iv) adaptation between memory areas; (v) adaptation between functional and internally generated interfaces.

The core of the distribution implementation is generated in interceptors called *RTZenStub* and *RTZenSkeleton* which mediate the communication with help of RTZen middleware. At the server side, *RTZenSkeleton* registers itself as a remote object in RTZen and serves like a proxy which delegates calls to a following interceptor which adapts an internally generated interface to the server component's functional interface. The *RTZenSkeleton* interceptor also configures a priority with which remote calls will be handled.

At the client side, *RTZenStub* obtains, via calling the encapsulated RTZen middleware, a reference to the remote object and delegates all incoming calls to it. However this reference implements the internally generated interface, therefore it has to be adapted to the functional interface by another interceptor called *Adaptor*. Finally, the *Serializer* interceptor arranges asynchronous semantics for method calls - each call on its provided interface is stored in a local queue and then served by a thread associated with the queue.

5 Conclusion and Future Work

Distribution support in RTSJ-based component systems is a highly demanded property which is however neglected in the state-of-the-art solutions. In this paper we propose an approach to introduction of distribution support into our framework for RTSJ systems [9]. We illustrate the ideas in a toy example introducing distribution using the RTZen middleware [10]. We show that our approach allows developers to hide the distribution support in container implementations which are automatically generated by our framework tool *Soleil*. As for the future work, we plan to conduct evaluation experiments measuring overhead of the framework. Similarly as we have shown in [9], the performance overhead should be minimal while preserving predictability. Moreover, we believe that our framework possesses a potential for a support of the `L2` level [12], we plan to investigate this opportunity.

References

[1] A. Plšek, P. Merle, L. Seinturier. A Real-Time Java Component Model. In *Proceedings of the 11^{th} International Symposium on Object/Component/Service-oriented Real-Time Distributed Computing (ISORC'08)*, pages 281–288, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[2] J. S. Anderson and E. D. Jensen. Distributed Real-Time Specification for Java: a Status Report (digest). In *Proceedings of the 4^{th} International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '06)*, pages 3–9, New York, NY, USA, 2006. ACM.

[3] P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Ayres. Towards the Integration of Scoped Memory in Distributed Real-Time Java. In *Proceedings of the 8^{th} IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '05)*, pages 382–389, 2005.

[4] A. Borg and A. Wellings. A Real-Time RMI Framework for the RTSJ. In *Proceedings of the 15^{th} Euromicro Conference on Real-Time Systems.*, pages 238–246, July 2003.

[5] T. Bures. *Generating Connectors for Homogeneous and Heterogeneous Deployment*. PhD thesis, Department of Sofware Engineering, Mathematical and Physical Faculty, Charles University, Prague, 2006.

[6] G. Bollela, J. Gosling, B. Brosgol, P. Dibble, S. Furr, M. Turnbull . *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[7] N. Medvidovic, E. M. Dashofy, and R. N. Taylor. The Role of Middleware in Architecture-Based Software Development. *International Journal of Software Engineering and Knowledge Engineering*, 13(4):367–393, 2003.

[8] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. In *Proceedings of the 22^{nd} International Conference on Software Engineering (ICSE '00)*, pages 178–187, New York, NY, USA, 2000. ACM.

[9] A. Plšek, F. Loiret, P. Merle, and L. Seinturier. A Component Framework for Java-based Real-time Embedded Systems. To Appear in *Proceedings of ACM/IFIP/USENIX 9^{th} International Middleware Conference*, Leuven, Belgium, December 2008.

[10] K. Raman, Y. Zhang, M. Panahi, J. A. Colmenares, R. Klefstad, and T. Harmon. RTZen: Highly Predictable, Real-Time Java Middleware for Distributed and Embedded Systems. In *Middleware 2005*, pages 225–248, December 2005.

[11] D. Tejera, A. Alonso, and M. A. de Miguel. RMI-HRT: Remote Method Invocation - Hard Real Time. In *Proceedings of the 5^{th} International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '07)*, pages 113–120, New York, NY, USA, 2007. ACM.

[12] A. Wellings, R. Clark, D. Jensen, and D. Wells. A Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation. In *Proceedings of ISORC '02.*, pages 13–22, 2002.